MASTER EN SCIENCES INFORMATIQUES

Intégration de mécanismes de chaînage avant dans un compilateur Prolog

Rybowski, Daniel

Award date:
1990

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Intégration de mécanismes de chaînage avant dans un compilateur Prolog

Daniel RYBOWSKI

Mémoire présenté en vue de l'obtention du titre
de Licencié et Maître en Informatique
Promoteur : Baudouin LE CHARLIER

Année académique : 1989-1990

Résumé

Une extension proposée à la programmation logique avec contraintes est l'intégration de mécanismes de chaînage avant permettant de "raisonner" sur les contraintes. Ces mécanismes peuvent en fait être appliqués à tout programme logique. Je décris ici la spécification de ces mécanismes et leur implémentation en un module intégrable à tout compilateur Prolog. Cette implémentation est basée sur la compilation des règles en chaînage avant en instructions d'une machine abstraite de Rete et sur l'émulation de cette machine. Une topologie particulière de réseau pour l'algorithme de Rete, facilitant la génération des instructions, est aussi examinée. Enfin, j'évoque quelques possibles applications de cette extension.

Abstract

A proposed extension for constraint logic programming is the integration of forward rules mechanisms, to be able to "reason" about the constraints. These mechanisms can actually be used for any logic program. I describe here the specification of these mechanisms as well as their implementation as a module that could be integrated in a Prolog compiler. These implementation is based on compiling the forward rules into instructions for a Rete abstract machine and a emulating such a machine. A particular topology for a Rete algorithm network, making the instructions generation easier, is also discussed. Finally, I discuss some possible applications for this extension.

TABLE DES MATIERES

CHAPITRE I : INTRODUCTION

CHAPITRE II : LE PROBLEME

CHAPITRE II : UNE SOLUTION AU PROBLEME POSE : LE LOGICIEL FR

CHAPITRE IV : "A QUOI CA SERT ?"

BIBLIOGRAPHIE

ANNEXE : SPECIFICATION DE L'INTERFACE DE FR

CHAPITRE I : INTRODUCTION

1.1. Prérequis : programmation logique et Prolog

La lecture fructueuse de ce mémoire requiert la connaissance de notions de programmation logique et de Prolog que l'on pourra acquérir, par exemple, dans (1).

1.2. Un problème pratique des langages de programmation logique : l'inefficacité temporelle

Grâce à sa nature non-déterministe et à sa forme relationnelle, la programmation logique est bien adaptée à la description de problèmes combinatoires, qu'elle permet d'exprimer sous forme de contraintes. Malheureusement, ce même caractère non déterministe est la cause d'une grande inefficacité temporelle de l'exécution des langages de programmation logique par des algorithmes à essais successifs.

1.3. Une tentative de résolution du problème de l'inefficacité temporelle : la programmation logique avec contraintes

La programmation logique avec contraintes est un développement récent, ayant comme objectif d'améliorer l'efficacité temporelle de la programmation logique tout en conservant la dualité de ses sémantiques déclarative et procédurale elle consiste à élaguer a priori l'ordre de recherche de l'algorithme à essais successifs en adjoignant à la programmation logique des domaines de calcul (tels les rationnels ou les booléens) et des mécanismes de résolution d'ensembles de contraintes sur ces domaines. (Pour un traitement en profondeur de la programmation logique avec contraintes, on pourra se reporter à (2). Les langages de programmation logique avec contraintes sont abordés notamment dans deux articles récents : (3) et (4).)

1.4. "Raisonner sur les contraintes" : la thèse de Thomas Graf

Dans sa thèse de doctorat (5), Thomas Graf propose une extension à la programmation logique avec contraintes : l'intégration de mécanismes de chaînage avant portant sur les contraintes. Il veut ainsi offrir un outil de "raisonnement" sur les contraintes, c'est à dire des techniques permettant au programmeur d'agir explicitement sur l'ensemble des contraintes, par exemple

pour en exploiter des propriétés globales, simplifier un système de contraintes ou déduire de nouvelles contraintes (redondantes) afin de réduire encore l'espace de recherche de l'algorithme à essais successifs. Il propose notamment des règles de production de deux types :

"- des règles de simplification qui permettent de remplacer un sous-ensemble de contraintes par un sous-ensemble équivalent, mais pouvant être résolu plus efficacement;

- des règles qui permettent, à partir d'un ensemble de contraintes, de déduire de nouvelles contraintes redondantes; celles-ci effectueront alors une réduction de l'espace de recherche qui ne serait pas possible en ne considérant que les contraintes initiales"(5). (Les règles de production sont décrites au paragraphe 2.3).

1.5. Contenu de ce mémoire

Ce mémoire décrit une implémentation de ces règles de production dans le cadre d'un compilateur pour CHIP, un langage de programmation logique avec contraintes dérivé de Prolog. Il est le résultat d'un stage que j'ai effectué d'août 1989 à janvier 1990 à l'European Computer Industry Research Center (ECRC), Münich, RFA.

1.6. Plan

Le chapitre II est consacré à la définition du problème à résoudre et aux spécifications décrites dans le cahier de charge initial. Le chapitre III examine les fonctionnalités du logiciel développé. Enfin, le chapitre IV discute des applications envisageables de ce logiciel.

1.7 Remerciements

Je remercie vivement le professeur Baudouin Le Charlier, promoteur de ce mémoire, pour ses critiques et remarques lucides. Je remercie également monsieur Pascal Van Hentenrijck qui me supervisa lors de mon séjour à Münich. Je désire aussi exprimer ma gratitude envers madame Françoise Berther, qui parvint à me supporter devant les six longs mois au cours desquels je partageai son bureau! Enfin, "last but not least", un grand merci aux chercheurs de l'ECRC pour leur chaleureux accueil.

CHAPITRE II : LE PROBLEME

2.1. Le problème à résoudre

On désire une implémentation des règles de production qui soit efficace au point de vue temporel, et si possible portable dans un autre contexte que celui du compilateur CHIP. La solution retenue fut un module compilant les règles de production en instructions d'une machine abstraite et émulant cette machine abstraite.

2.2. Cahier de charges initial

Les spéculations initiales pourraient être paraphrasés comme suit : "On demande d'implémenter des règles de production par une machine abstraite pour l'algorithme de Rete (voir § 2.4), en se basant sur le jeu d'instructions conçue par Thomas Graf pour une première implémentation dans le cadre d'un interpréteur pour CHIP. Cette implémentation comprendra un compilateur des règles de production en instructions de la machine abstraite et un émulateur de cette machine abstraite. Elle devra prendre la forme d'un module utilisable par le compilateur CHIP développé à l'ECRC et plus généralement par une gamme aussi large que possible de compilateurs pour Prolog et ses dérivés. Il faut que le compilateur CHIP soit capable de traiter les règles de production simplement en éxécutant une procédure "built-in" (qui invoquera le compilateur de règles de production) et en ajoutant dans son code généré (là où approprié) un appel de l'émulateur de règles de production. Le logiciel devra être écrit en langage C".

Nous examinerons ci-dessous les notions nécessaires à la compréhension de ces spécifications.

2.3. Les règles de production (5)

2.3.1. Syntaxe (abstraite)

Dans notre contexte, une règle de production est une construction d'une des deux formes suivantes :

Règle de simplification : $L_1\ L_2 \ldots\ L_m \Leftrightarrow R$

Règle d déduction : $L_1\ L_2 \ldots\ L_m \Leftrightarrow R$

où les Li ($1 \leq i \leq m$) sont des termes Prolog. Afin de simplifier l'écriture du compilateur, les L_i furent limités à des constantes et à des structures dont les arguments sont des constantes ou des variables . Ces restrictions pourraient être levées si désiré. D'autre part, R peut être soit un terme Prolog, soit le prédicat d'échec fail/0, soit le prédicat d'unification =/2. Dans ce dernier cas, il faut que l'un des arguments de =/2 soit le nom d'une variable apparaissant comme argument d'un des L_i et que l'autre argument soit le nom d'une autre telle variable ou une constante.

2.3.2. Explication intuitive et exemples

Une règle de simplification L_1 & L_2 & ... & L_m $\Leftrightarrow$ R a la lecture déclarative "L_1 et L_2 et...et L_m est équivalent à R" et la lecture procédurale "remplacer le sous-ensemble de contraintes { C_1, C_2,...C_m } par R si { C_1, C_2,...C_m } = { $L_1\theta$ $L_2\theta$,...,$L_m\theta$ }

- $x \geq 0$ & $x \geq 1$ $\Leftrightarrow$ $x \geq 1$ signifie au point de vue déclaratif
" $x \geq 0$ et $x \geq 1$ est équivalent à $x \geq 1$" et au point de vue procédural :
"remplacer les contraintes { RES≥0, RES≥1 } par { RES≥0 }"

- $x \geq 0$ & $x \leq 0$ $\Leftrightarrow$ $x = 0$ signifie au point de vue procédural :
"remplacer les contraintes { N≥0, N≤0} par l'unification de N à 0"

-$x \geq 0$ & $x \leq 0$ $\Leftrightarrow$ fail signifie au point de vue procédural :
"les contraintes { K≥0, K<0 } sont contradictoires : il faut effectuer un backtratching et supprimer ces contraintes"

Une règle de déduction L_1&L_2& ... & L_m R a la lecture déclarative "L_1 et L_2 et ....et L_m implique R" et la lecture procédurale "Si l'ensemble des contraintes contient { C_1, C_2,..., C_m } = { $L_1\theta$, $L_2\theta$,..., $L_m\theta$ }, y ajouter $R\theta$".

- $x \geq y$ & $x \geq y$ $\Leftrightarrow$ $x \geq z$ signifie au point de vue déclaratif : " si $x \geq y$ et si $y \geq z$ alors $x \geq z$" et au point de vue procédural :
" si { R1≥R2, R2≥R3 } est inclus dans l'ensemble des contraintes S,
S $\leftarrow$ S $\cup$ { R1≥R3}".

2.3.3. Sémantique

Une règle de production est applicable par rapport à un ensemble de contraintes S si et seulement si il existe un sous-ensemble de contraintes $\{ C_1, C_2, ..., C_m \} \subset S$ et une substitution θ telles que :

$i : 1 \leq i \leq m : C_i = L_i\theta \quad (m \geq 1)$

L'application des règles applicables modifie S : soient R_s l'ensemble des règles de simplification applicables par rapport à S, R_d l'ensemble des règles de déduction applicables par rapport à S :

$R_s = \{ r_{s1}, r_{s2}, ..., r_{sn} \}$, $R_d = \{ r_{d1}, r_{d2}, ..., r_{dp} \}$

- $R_s = \{ \emptyset \ (n=0) \Leftrightarrow S \leftarrow S \cup \{ \overrightarrow{r_{d1}}\theta_1, \overrightarrow{r_{d2}}\theta_2, ..., \overrightarrow{r_{dp}}\theta_p \}$ où $\theta_1, \theta_2, ..., \theta_n$ sont les substitutions grâce auxquelles $r_{d1}, r_{d2}, ..., r_{dp}$ sont applicables et $\overrightarrow{r_{d1}}$, $\overrightarrow{r_{d2}}, ..., \overrightarrow{r_{dp}}$ sont les membres de droite des règles.

- $R_s \neq \{ \emptyset \} \ (n>0) \Leftrightarrow S \leftarrow (S \setminus \{ \overleftarrow{r_{si}}\theta_i \}) \cup \{ \overrightarrow{r_{si}}\theta_i \}$ où $\overleftarrow{r_{si}}$ et $\overrightarrow{r_{si}}$ sont respectivement le membre de gauche et de droite de la règle r_{si} $(1 \leq i \leq n)$ et θ_i la substitution pour laquelle r_{si} est applicable, i étant choisi de manière quelconque.

2.4. Un modèle d'éxécution des règles de production: l'algorithme de Rete (6)

2.4.1. Spécification fonctionnelle du modèle d'éxécution des règles de production

Soient S un ensemble de contraintes, C une contrainte (n'appartenant pas nécessairement à S), R un ensemble de règles de production :

Préconditions : /

Postconditions : les règles de production $\in$ R applicables par rapport à S U { C } et non applicables par rapport à S ont été appliquées. (Notons que rien n'est spécifié au sujet des autres règles applicables).

Il s'ensuit que, pour appliquer les règles de production pour une suite de contraintes $C_1, C_2, ..., C_m$ $(m \geq 0)$, il suffit que le logiciel utilisant le module d'éxécution des règles de production l'invoque dès l'apparition d'un nouveau C_i $(1 \leq i \leq m)$ avec $S = \{ C_1, C_2, ..., C_{i-1} \}$.

2.4.2. Mise en oeuvre de la spécification

Une approche naïve consiste à rechercher l'applicabilité de toutes les règles par rapport à toutes les contraintes lors de chaque invocation du module. Bien entendu, cette approche est d'une grande inefficacité temporelle.

Comme le remarque Thomas Graf, la spécification ci-dessus est similaire à celle de systèmes de règles de production tels OPS-5 si nous considérons les contraintes comme les éléments de la mémoire de travail. Les hypothèses sur lesquelles reposent les algorithmes de "pattern-matching" utilisés par ces systèmes sont aussi vérifiées dans notre cas.

Ces hypothèses sont :

- (quasi-) invariabilité des règles durant l'éxécution, ce qui rend attractive une compilation de ces règles;
- faible nombre de modifications de l'ensemble des contraintes par rapport à la taille de cet ensemble; il est donc moins coûteux de mémoriser et de mettre à jour l'état de la mémoire de travail que de le recalculer entièrement à chaque activation du module;
- un même littéral peut se retrouver dans le membre de gauche de plusieurs règles; il est alors intéressant d'éviter des traitements redondants.

Un des algorithmes répondait à ces hypothèses est l'algorithme de Rete, qui a été choisi (cf. § 2.2).

2.4.3. Algorithme de Rete : introduction

Cet algorithme comprend deux phases :

1) la compilation des membres de gauche des règles de production en un réseau dont les noeuds mémoriseront l'état de la mémoire de travail, c'est -à -dire l'état d'instanciation des règles (voir § 2.4.4.).

2) le parcours de ce réseau pour chaque nouvelle contrainte afin de mettre à jour l'état de la mémoire de travail et éventuellement de signaler qu'une ou plusieurs règles sont désormais applicables (voir § 2.4.5.).

2.4.4. Réseau de Rete

Que signifie l'expression "mémoriser l'état d'instanciation des règles" ? Soit une règle de production $L_1 \& L_2 \& ... \& L_m \rightarrow R$ où $\rightarrow$ est soit $\Rightarrow$, soit $\Leftrightarrow$.

Soit une contrainte $C_1 = L_i\vartheta_1$ ($1 \leq i \leq m$). Nous pourrions mémoriser dans un noeud consacré à L_i la substitution ϑ_1. Supposons encore que survienne par la suite une contrainte $C_2 : L_j\vartheta_2$ ($1 \leq j \leq m$, $j \neq i$). Nous pouvons alors mémoriser dans le noeud de L_j la substitution ϑ_2. Nous pourrions aussi aller plus loin et mémoriser dans un noeud du réseau de l'instanuation "partielle" $(L_i \& L_j)\vartheta_1\vartheta_2 = C_1 \& C_2$ (à condition que ϑ_1 et ϑ_2 ne soient pas contradictoires, c'est à dire qu'ils ne substituent pas une même variable X deux constantes distinctes n_1 et n_2). Notons tout de suite que le noeud $L_i \& L_j$ n'existe pas nécessairement, grâce aux propriétés d'associativité et de commutativité de : par exemple, notre réseau pourrait contenir un noeud $L_k \& L_j$ ($1 \leq k \leq m$, $k \neq i$, $k \neq j$) et un autre noeud $L_k \& L_j \& L_i$. Il ne serait pas possible alors de joindre L_i et L_j avant d'avoir instancié "partiellement" L_k (Nous examinerons le problème de la topologie du réseau au § 3.4.1.).

Cet exemple nous permet de distinguer trois types de noeuds :

1) les noeuds "élémentaires", mémorisant les contraintes (et les substitutions) instanciant un littéral dans le membre de gauche d'une règle. A noter que, quel que soit le nombre de membres à gauche où ce littoral apparaît, il suffit d'un seul noeud élémentaire ayant plusieurs descendants.

2) les noeuds "de jointures mémorisant les instanciations partielles d'un "morceau" de membre de gauche.

3) les noeuds "terminaux" correspond à l'instanciation complète d'un membre de gauche, c'est-à-dire aux règles applicables. Notons que si m=1, le noeud élémentaire est terminal.

Remarquons que chaque noeud de jointure ou terminal a deux pères : nous les appellerons "père de gauche" et "père de droite". Par exemple, le noeud associé à $L_k \& L_j$ a pour "père de gauche" le noeud de L_k et pour "père de droite" le noeud L_j.

Résumons notre propos d'une manière quelque peu plus abstraite : un réseau de Rete est un graphe orienté. Les noeuds mémorisent les substitutions permettant d'instancier partiellement le membre de gauche $L_1 \& L_2 \& \ldots \& L_m$ d'une règle de production. Les arcs déterminent les chemins parcourus lors de l'éxécution de l'algorithme décrit au §2.4.5. Cet algorithme a pour but de répercuter toutes les conséquences de la survenance d'une nouvelle contrainte. Pour ce faire, il tente la jointure de tous les nouveaux éléments d'un "père de gauche" avec les éléments contenus dans "le père de

droite" correspondant (ou réciproquement) : chaque jointure réussie entraîne l'ajout d'un nouvel élément dans le noeud "fils". Si ce noeud "fils" est lui-même "père" d'un autre noeud (autrement dit, s'il correspond à un "morceau" de règle, autrement dit encore, s'il est un noeud de jointure), l'algorithme en tire les nouvelles conséquences récursivement; si le noeud "fils" n'a pas de "fils", c'est donc qu'il est un noeud terminal, correspondant au membre de gauche d'une règle de production, et les nouveaux éléments représentent des instanciations complètes de cette règle désormais applicable.

2.4.5. Algorithme abstrait de parcours d'un réseau de Rete

2.4.5.1. Spécification fonctionnelle

Soient C une contrainte, N un réseau de Rete, $N1_i(0 \leq j \leq m)$ les noeuds élémentaires de ce réseau, N_{2j} $(0 \leq j \leq m)$ les noeuds de jointures, N_{3k} $(0 \leq k \leq p)$ les noeuds terminaux N_{4l} $(0 \leq l \leq q)$ les noeuds élémentaires terminaux.

Précondition : $\{ N1_i \forall i : 1 \leq i \leq m, N_{2j} \forall j : 1 \leq j \leq n \}$ mémorise un état (éventuellement vide) d'instasuation partielle des règles de production compilées, correspondant à un certain ensemble de contraintes S.

Postcondition : $\{ N1_i \forall i : 1 \leq i < m, N_{2J} \forall j : 1 \leq j \leq n \}$ mémorise un état d'instanciation partielle des règles de production compilées, correspondant à S U { C} et les règles de production correspondant aux N_{3k} et N_{4l} faisant l'objet d'une instanciation complète sont applicables.

2.4.5.2. Algorithme

L'algorithme de parcours consiste à mettre à jour les noeuds dès qu'une nouvelle contrainte est introduite. Il comprend quatre phases :

1) la phase de filtrage consiste à vérifier l'instanciation partielle d'un littéral L. Si c'est le cas, la substitution $\theta : C = \theta$ approprié est mémorisé dans le noeud élémentaire correspondant à ce littéral et on passe à la phase 2, sinon, on passe à la phase 4.

2) la phase de jointure consiste à vérifier l'instanciation partielle d'un "morceau" de règle de production; si une instanciation partielle réussit alors on passe à la phase 3 ou 2 selon que le "morceau" de règle que l'on vient d'examiner était ou non la règle entière; sinon, on passe à la phase 4.

3) une règle est applicable (on est au niveau d'un noeud terminal).

4) traitement du cas des noeuds élémentaires terminaux.

Structures de données : on suppose ci-dessous que les noeuds non terminaux sont des ensembles de substitutions θ.

L'algorithme est exprimé dans un pseudo-code informel. A ma connaissance, cet algorithme abstrait n'avait pas encore été publié.

```
Rete (C,N)
    Pour tout noeud élémentaire N1i
        { phase 1 }
        Si C est une instance du littéral Li associé à N1i alors
          soit θ la substitution correspondante : N1i ← N1i ∪ { θ }
                                                  S ← { θ }
                                                  { S a la même
                                                  structure que les
                                                  noeuds non
                                                  élémentaires }
            { phases 2 et 3 }
            fin si phase 2 -3 (S,N1i,N)
        fin pour
    Pour tout noeud élémentaire N4l
        { phase 4 }
        Si C est une instance du littéral Ll associé à N4l alors
          soit θ la substitution correspondante :
                " les règles de production dont le membre de gauche
                est Ll sont applicables avec la substitution θ "
        fin si
    fin pour
    retourner N
fin
```

```
phase 2-3 (S1, P, N)
        { S1 et S2 ont la même structure que les noeuds non
         terminaux, P est un noeud non terminal, S1 contient
         l'ensemble des substitutions de P n'ayant encore fait l'objet
          d'aucune tentative de jointure }
       S2 ← { ø }
        Pour tout noeud fils de P : soit F
              soit P2 le deuxième père de F
              Pour tout θ 1∈S1
                 Pour tout θ 2∈P2
                       Si θ 1 et θ 2 ne sont pas contradictoires
                    (cf.2.4.4.) alors
                          si P est le père de gauche de F alors
                            S2 ← S2 U { θ 1θ 2 }
                          sinon
                            S2 ← S2 U {θ 2θ 1 }
                          fin si
                       fin si
                 fin pour
              fin pour
              Si F est un noeud terminal ( k : F = N3k) alors
                 { phase 3 }
                 Pour tout θ∈S2
                       " les règles de production dont le membre de
                        gauche est associée au noeud F sont
                        applicables avec la substitution θ "
                 fin pour
              sinon { F est un noeud de jointure : j : F = N2j; F a un
                       fils }
                    { phase 2 }
                    F ← F U S2
                     phase 2-3 (S2, F, N)
              fin si
           fin pour
           retourner N
fin
```

CHAPITRE III :
UNE SOLUTION
AU PROBLEME POSE :
LE LOGICIEL FR

3.1. Structure et composants de FR

FR est structuré comme un module offrant à l'utilisateur 2 fonctions (en fait 3, voir end_compile () en annexe) : le compilateur de règle de productions : compiler () et l'émulateur permettant d'éxécuter les règles compilées : emulator (). Il requiert en outre l'écriture d'un interface spécifique à l'utilisateur (cet interface est complètement décrit en annexe).

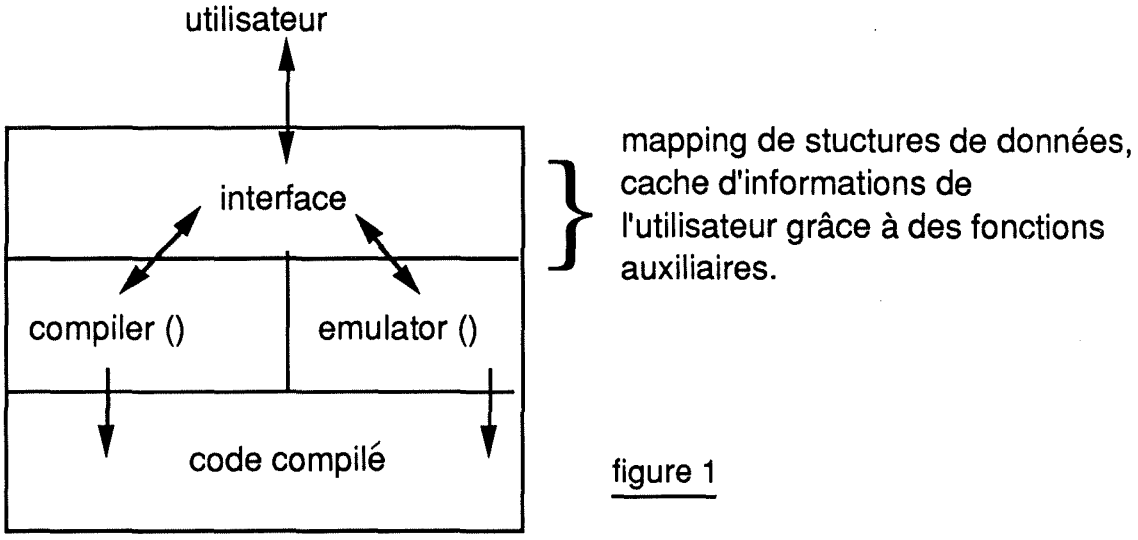


figure 1

L'utilisateur n' a accès aux fonctions que via l'interface. Les fonctions rendent leurs résultats à l'utilisateur via l'interface.

3.2. Adaptations de l'algorithme de Rete

Par rapport à l'algorithme tel qu'esquissé au chapitre précédent, il nous faut accomoder :

a) la présence de termes structurés (à arguments simples).

b) la présence de variables dans les contraintes : leur instanuation pourrait rendre de nouvelles règles applicables, il faut donc reconsidérer la contrainte à chaque instanciation de variable. Par exemple, soit S = { p(X,3) } si X était instancié à 8, la règle p(8,3) $\Rightarrow$ fail serait applicable.

c) la suppression de contraintes : dans l'algorithme de Rete, la suppression d'un élément de la mémoire de travail entraîne une mise-à-jour explicite de l'état d'instanciation des règles. Dans FR, les contraintes sont toujours supprimées implicitement, soit qu'elles soient "entièrement résolues" par le résolveur de contrainte du langage de programmation logique avec

contraintes, soit qu'elles soient victimes de l'application d'une règle de simplification (voir § 2.3.).

3.3. Machine abstraite de Rete

3.3.1. Introduction

Un réseau de Rete est implémenté par les instructions d'une machine abstraite. Nous allons décrire ci-dessous une telle machine abstraite. Ses instructions sont fort proches de celles définies par Thomas Graf pour son implémentation initiale des règles de production.

3.3.2. Jeux d'instructions

Nous distinguerons 5 types d'instructions :

- instructions de filtrage, correspondant à la compilation de la phase de filtrage de l'algorithme ce sont les instructions get_atomic get_value et store_cstr.
- instructions de jointure correspondant à la phase de jointure : left merge, right merge, remerge, combine, test_eq.
- instructions relatives aux règles applicables : terminal, terminal0
- instruction navigation dans le réseau : fork, stop
- instructions relatives à l'application d'une règle (compilation de la partie droite) : make_goal, push_atomic push_var, push_value, push_ref, unifly_atomic, unify_value, succeed_all, return, fail.

Chaque instruction est composée d'un code opératoire et de 0,1,2,3 ou 4 arguments.

3.3.2.1. Structures de données

La machine abstraite opère sur des piles de noeuds. Les éléments des noeuds élémentaires ont la structure logique suivante :

référence à une contrainte.

Les éléments des noeuds de jointure et des noeuds terminaux ont la structure logique suivante :

référence à la 1ère contrainte
référence à la 2ème contrainte

.
.
.

La machine abstraite dispose aussi d'une pile d'adresses d'instructions P et des registres suivant : C: nouvelle contrainte à traiter

A : adresse

PN1,PN2 : identifiants de piles de noeuds

DIR : { true , false }

I : entier

3.2.2.2. Les instructions

GET_ATOMIC : 2 arguments : arg_1 un atomique (c'est-à-dire un atome, une constante numérique ou une chaîne de caractères) et arg_2 un numéro d'argument de la contrainte contenue dans le registre C.

Pré C contient une structure d'arité m>0; $0<arg_2 \leq m$;

pile d'adresses = adr_n $n \geq 0$

.

.

.

adr_1 $n \geq 0$

Post - si l'argument en position arg_2 de C (noté $C(arg_2)$) = arg_1, passage à l'instruction suivante.

- si $C(arg_2) \neq arg_1$ alors :

* si n>0, pile d'adresses = adr_{n-1}

.

.

.

adr_1

et l'éxécution se poursuit à l'adresse adr_n

* si n=0, l'éxécution prend fin

GET_VALUE : identique à GET_ATOMIC si ce n'est que la comparaison porte sur les arguments de C : $C(arg_1)$ et C (arg_2) (arg_1 et arg_2 sont des numéros d'arguments de C). Cette instruction sert à vérifier l'identité de 2 arguments d'un même littéral : par exemple, si p(X,Y,X) est un littéral d'une règle, l'identité du 1er et du 3ème argument serait vérifiée par une instruction GET_VALUE 1,3.

STORE_CSTR un argument arg_1 = l'identifiant d'une pile de noeuds élémentaires (notée $Parg_1$)

Pré $Parg_1 = n_m$ où ni est la référence à une contrainte, $m \geq 0$

n_{m-1}

.

.

Post : P arg_1 = réf C où réf C est la référence à la contrainte C

n_m
.
.
.
n_2
n_1

TERMINALQ : identique à STORE -CSTR; cette instruction correspond au cas spécial d'un membre de gauche ne comprenant qu'un seul littéral (voir §2.4.3.)

LEFT-MERGE : 2 arguments : arg_1 et arg_2 sont des identifiants de piles de noeuds.

Post : $PN_1 = arg_1$, $PN_2 = arg_2$, DIR = true

RIGHT-MERGE : 3 arguments : arg_1 et arg_2 sont des identifiants de piles de noeuds, arg_3 est l'adresse d'une instruction.

Pré : P arg_1= n_{1m} P arg_2 = n_{2m}

n_{1m-1} n_{2m-1}
. .
. .
. .
n_{1i} n_{2i}

où n_{ij} est un élément de noeud.

Post : $PN_1 = arg_1$, $PN_2 = arg_2$, DIR = false

Si m>0 et $n_{1m} = n_{2m}$ alors P arg_1 = n_{1m-1}

.
.
.
n_{i1}

Parg_2 est inchangé

L'éxécution se poursuit en arg_3.

TEST_EQ : 3 arguments : arg_1 ~~et arg_2~~ est un numéro de littéral d'un membre de gauche d'une ~~même~~ règle de production, arg_2 et arg_3 sont des numéros d'arguments de ces littéraux arg_1 et L, où L est le littéral instancié par C.

Pré $1 \leq arg_1 \leq$ nombre de littéraéux de la règle;

le littéral représenté par arg1 n'est pas L;

$1 \leq arg_2 \leq$ arité du littéral arg_1;

$1 \leq arg_3 \leq$ arité de L

$P_{Pn1} = n_{1m}$ $P_{Pn2} = n_{2n}$ $m,n \geq 0$

n_{1m-1} n_{2n-1}

. .

. .

. .

n_{11} n_{21}

PPN2 est une pile de noeuds élémentaires

Pile d'adresses = adr_p $p \geq 0$

.

.

.

adr_1

$I = I_0$, $DIR = DIr_0$

Post Soit b la arg_1 ème référence à une contrainte dans $n_{1m'}$, avec $m' = m$ si DIR_0 = true, $m\text{-}I_0$ sinon;

$n' = n$ si DIR_0 = false, $n\text{-}I_0$ sinon

- Si $m'>0$ et $n'>0$ et $b(arg_2) = n_{2n'}$ (arg_3) et les contraintes référencées par b et $n_{2n'}$ n'ont pas été supprimées alors l'éxécution se poursuit séquentiellement

- sinon, * si $p>0$, l'éxécution se poursuit en adr_p et

pile d'adresses = adr_{p-1}

.

.

.

adr_1

* sinon, l'éxécution prend fin

REMERGE : pas d'argument

Post A = l'adresse de l'instruction suivante, I = 0

COMBINE : un argument arg_1 identifiant une pile de noeuds de jointure

Pré $Parg_1 = n_{1m}$ $P_{PN1} = n_{2n}$ $P_{PN2} = n_{3p}$

. . .

. . .

. . .

n_{11} n_{21} n_{31}

$m \geq 0$, $I = I_0$, $DIR = DIR_0$

Si DIR_0 = true , $m' = m > 0$, $p' = p - l_0 > 0$

Si DIR_0 = false, $n' = n - l_0 > 0$, $p' = p > 0$

Post $Parg_1 = n_{1n+1}$ avec $n_{1m+1} = ref\ but_{k+1}$ } $n_{3p'}$

n_{1m} $ref\ but_k$

. . .

. . $n_{2n'}$

. .

n_{11} $ref\ but_{1+1}$

$l = l_0+1$

Si (DIR0 = true et l<p) ou (DIR0 = false et l<m)

alors l'éxécution se poursuit en A

sinon l'éxécution se poursuit séquentiellement

TERMINAL : identique à COMBINE si ce n'est qu'arg_1 identifie un noeud terminal

MAKE GOAL : 2 arguments : arg_1 est le foncteur d'un terme Prolog à créer et arg_2 est son arité

Pré $arg_2 \geq 0$

Post la demande de création d'un terme de foncteur arg_1 et d'arité arg_2 est prête, les arguments ne sont pas initialisés; l = 1

PUSH_ATOMIC : un argument arg_1 atomique

Pré $l = l_0$

Post : le l_0 ème argument du nouveau terme est arg_1; $l = l_0+1$

PUSH_VAR : pas d'argument

Pré $l = l_0$

Post le l_0 ème argument du nouveau terme est une variable non instanciée; $l = l_0 + 1$

PUSH-VALUE : 2 arguments : arg_1 = un numéro de littéral du membre de gauche d'une règle de production, arg_2 = un numéro d'argument de ce littéral

Pré $l = l_0$, $1 \leq arg_1 \leq$ nombre de littéraux du membre de gauche,

$1 \leq arg_2 \leq$ arité du littéral arg_1

Post le $l_0$ème argument du nouveau terme est une variable instanciée au arg_2ème argument de la arg_1ème composante du sommet du noeud terminal courant, $l = l_0 + 1$

PUSH-REF : un argument arg_1 = un numéro d'argument du nouveau terme Prolog

Pré $l = l_0$, $1 \leq arg_1 < l_0$

Post : le $l_0$ème argument du nouveau terme est une variable instanciée au arg_1 ème argument de ce but, $l = l_0 + 1$

UNIFY_ATOMIC : 3 arguments : arg_1 un atomique, arg_2 un numéro de littéral du membre de gauche d'une règle de production, arg_3 un numéro d'argument de ce littéral

Pré $1 \leq arg_2 \leq$ nombre de littéraux du membre de gauche,

$1 \leq arg_3 <$ arité du littéral arg_2

Post demander l'unification d'arg_1 et de la variable arg_3ème argument de la arg_2ème composante du sommet du noeud terminal courant

UNIFY_VALUE : 4 arguments : arg_1 et arg_3 sont des numéros de littéraux du membre de gauche d'une même règle de production, arg_2 et arg_4 sont des numéros d'arguments de ces littéraux

Pré $1 \leq arg_1, arg_3 \leq$ nombre de littéraux du membre de gauche,

$1 \leq arg_2 \leq$ arité du littéral arg_1,

$1 \leq arg_4 \leq$ arité du littéral arg_3

Post demander l'unification du arg_2ème argument de la arg_1ème composante et du arg_4ème argument du arg_2ème composante du sommet du noeud terminal courant

SUCCEED-ALL : pas d'argument

post demander que les contraintes composantes du noeud du noeud terminal courant soient supprimés .

RETURN : pas d'argument

pré pile d'adresses = adr_1

.

.

.

adr_m $(m \geq 0)$

post Si m>0, pile d'adresses = adr_1

.

. et l'éxécution se poursuit

. en adr_m

adr_{m-1}

Sinon, l'éxécution prend fin.

FAIL : pas d'argument

Pré pile d'adresses = adr_1

.

.

adr_m $(m \geq 0)$

Post : demander l'adjonction d'un fail/0 dans l'ensemble des contraintes de l'environnement Prolog; si m>0, pile d'adresses =

adr_1

.

.

.

adr_{m-1}

et l'éxécution se poursuit en adr_m, sinon l'éxécution prend fin.

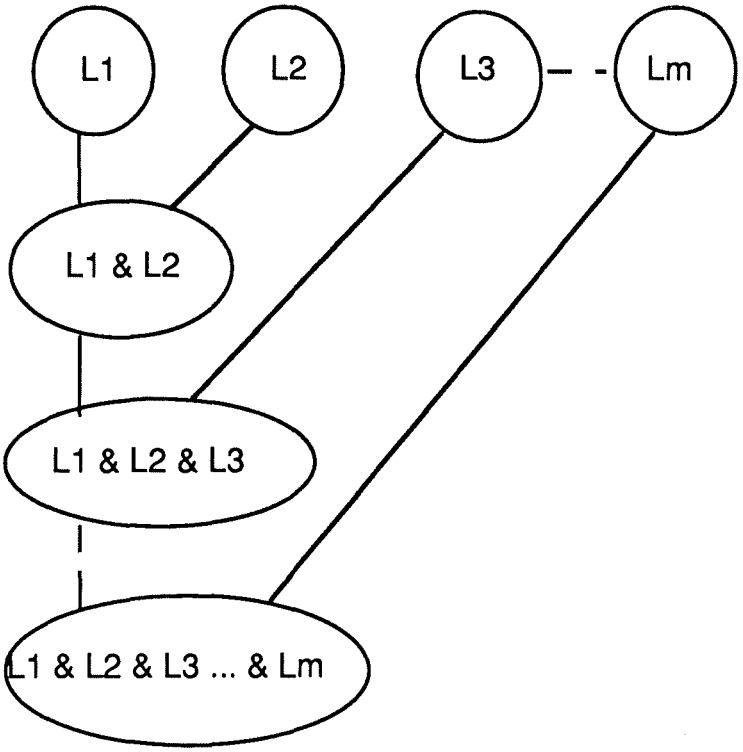
STOP : pas d'argument

Post : l'éxécution prend fin.

3.4. Le compilateur

3.4.1. Topologie du réseau de Rete

Combien de noeuds un réseau de Rete comprend-t-il au minimum ? pour nous faciliter la tâche, considérons qu'il n'y a qu'une seule règle L_1 & L_2&.... &...&. $L_m \Leftrightarrow R$ à compiler. Clairement nous aurons m noeuds élémentaires (si tous les Li sont distincts). Nous aurons au moins m-1 noeuds de jointure ou terminaux



m-1 noeuds non élémentaires

figure 2

Il serait redondant d'ajouter par exemple un noeud de jointure entre L2 et L3:

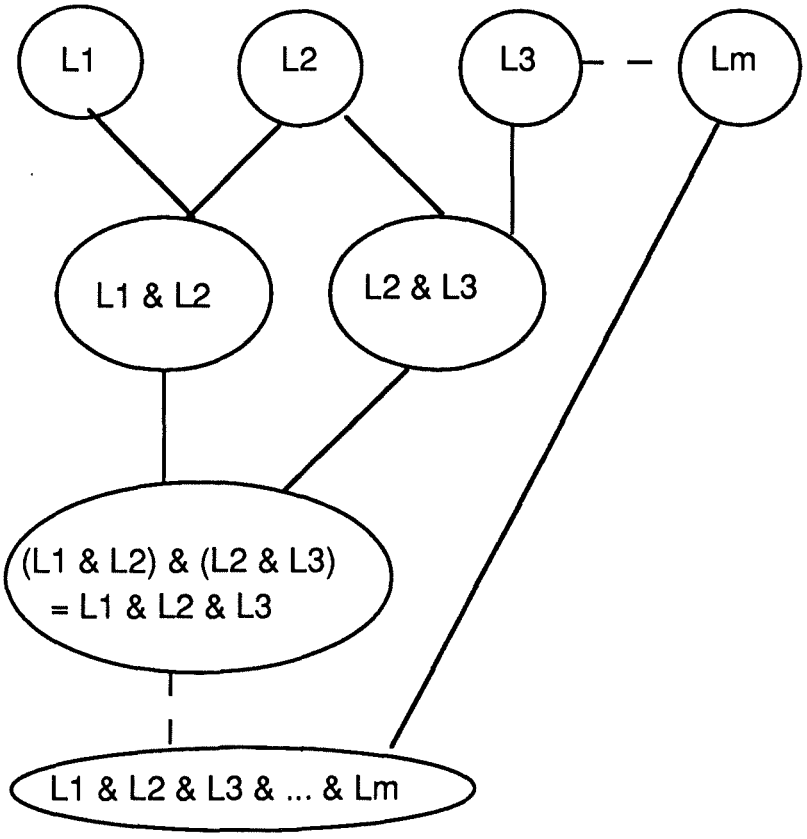


figure 3

(m-1) +1 = m noeuds non élémentaires

Cette redondance existe non seulement au point de vue spatial (nombre de noeuds), mais aussi au point de vue temporel : on effectue une jointure de plus sans arriver plus vite au résultat final (le noeud terminal) pour lequel nous avons de contraintes instanciant tous les littéraux.

Cependant, plusieurs topologies minimisent le nombre de noeuds, toutes celles en fait où chaque littéral n'intervient tout seul que dans une et une seule jointure :

par exemple :

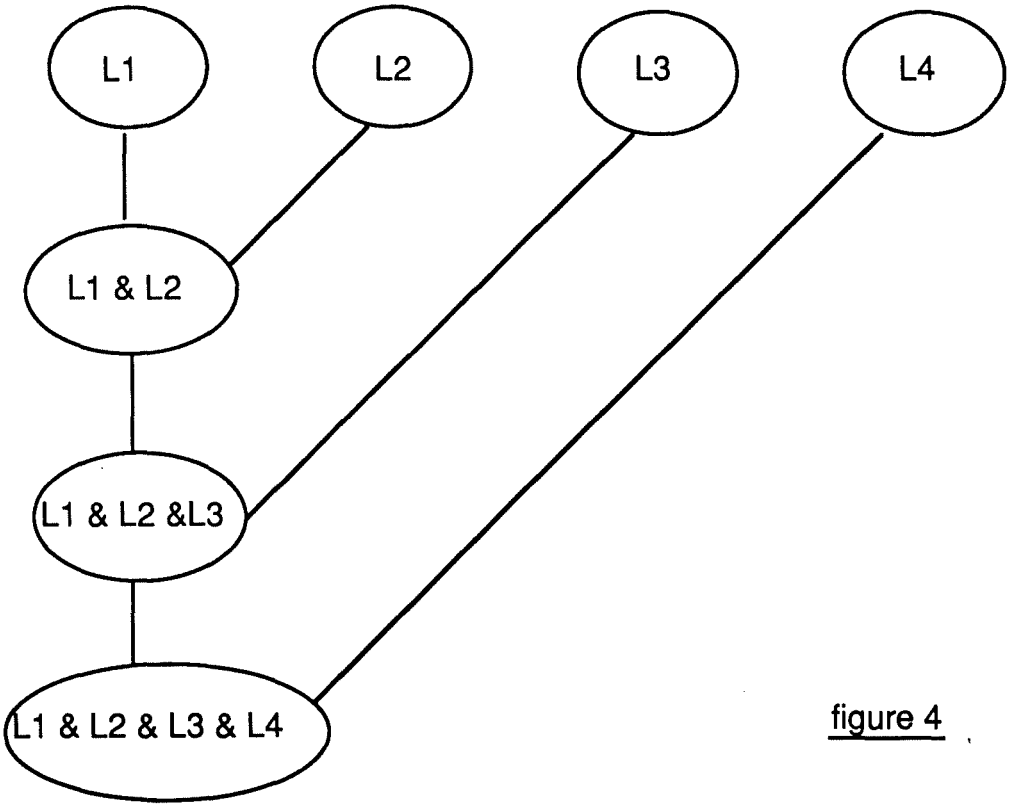


figure 4

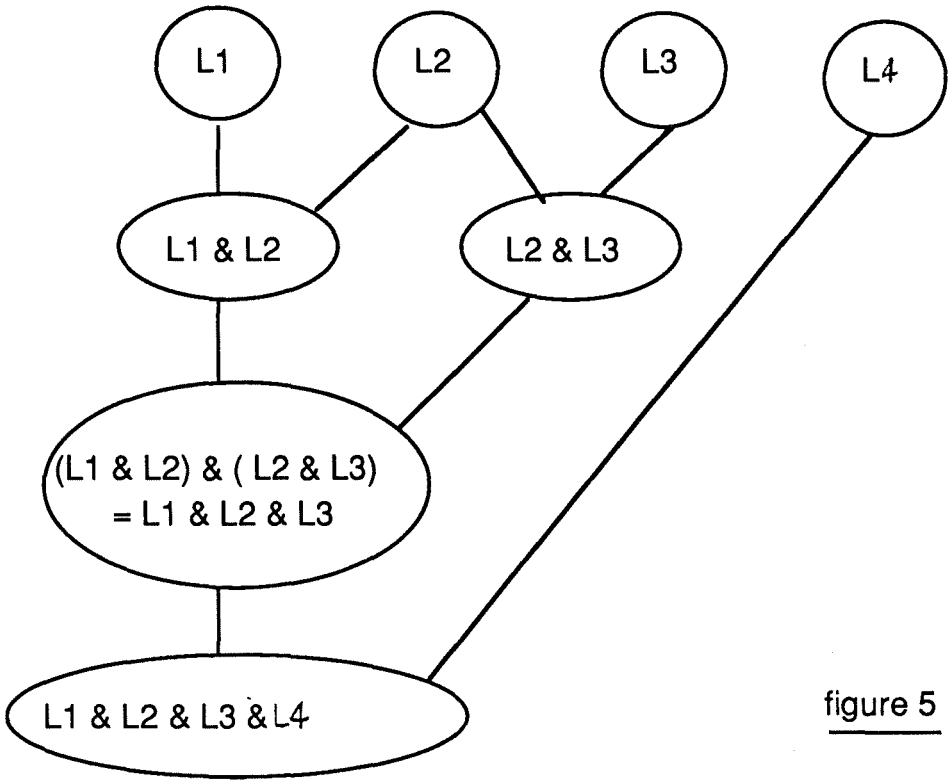


figure 5

ou encore :

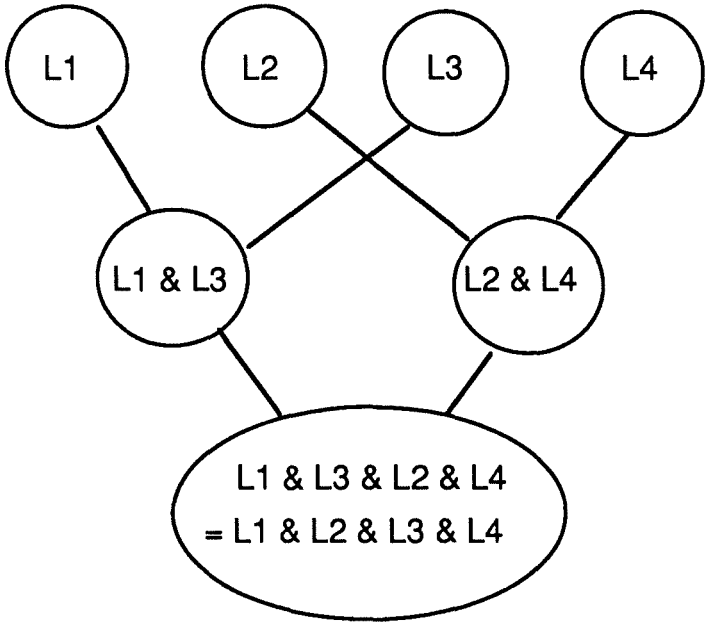


figure 6

En fait, toutes les topologies utilisant les propriétés d'associativité et de commutativité de & sont équivalentes en nombre de noeuds.

Le choix s'est finalement porté sur l'évaluation progressive des jointures de gauche à droite (comme dans l'exemple de la figure 4) Deux raisons à cela :

1) on peut s'attendre à ce que le programmeur mette les contraintes les plus fortes à la gauche de la règle; le nombre de jointures partielles serait alors minimisé, et donc aussi la quantité d'espace-mémoire dévolue aux noeuds de jointure

2) ce traitement de gauche à droite se prête très bien à la génération des instructions (voir § 3.4.2).

3.4.2. Génération des instructions

Cette génération des instructions se fait conformément à la topologie choisie. Aucune considération quant à l'éxécution des instructions n'a place ici : il nous suffit de montrer que le code généré est bien une représentation des noeuds et des arcs d'un réseau de Rete, l'exploitation de ce réseau est à charge de l'émulateur. Dès lors, il nous suffit de distinguer 2 cas :

1) compilation d'un membre de gauche composé d'un seul littéral L : il nous faut représenter l'arbre "dégénéré" (L) noeud élémentaire terminal.

Code à générer : instructions de filtrage (get-atomic et get-value)

terminal0

2) compilation d'un membre de gauche composé d'au moins 2 littéraux :

Si nous examinons plus attentivement l'exemple de la figure 4 du § 3.4.1., nous constatons que chaque noeud de jointure vient s'ajouter à une file de noeuds entamée par le noeud de filtrage du littéral le plus à gauche. Nous allons exploiter cette propriété en générant une séquence d'instruction entamée par le filtrage du littéral le plus à gauche et se poursuivant par les instructions représentant l'arc de gauche d'une jointure (left-merge) et le noeud de jointure lui-même. (Notons que pour cette raison left-merge ne comprend pas d'adresse d'instruction en 3ème argument à la différence de son homologue right-merge).

Nous pouvons distinguer trois cas :

a) littéral le plus à gauche :

code à générer : instructions de filtrage

L_1 left-merge } S

b) littéral autre que le premier ou le dernier:

code à générer : - compléter S par tests de jointure

COMBINE

LEFT_MERGE (représente l'arc de la jointure suivante)

Li REMERGE

- générer : instructions de filtrage du littéral

RIGHT_MERGE L_{i-1}

c) dernier littéral : au lieu d'un noeud de jointure, nous avons besoin d'un noeud terminal

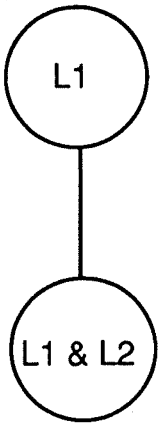
code à générer : - compléter S par : test de jointure

TERMINAL

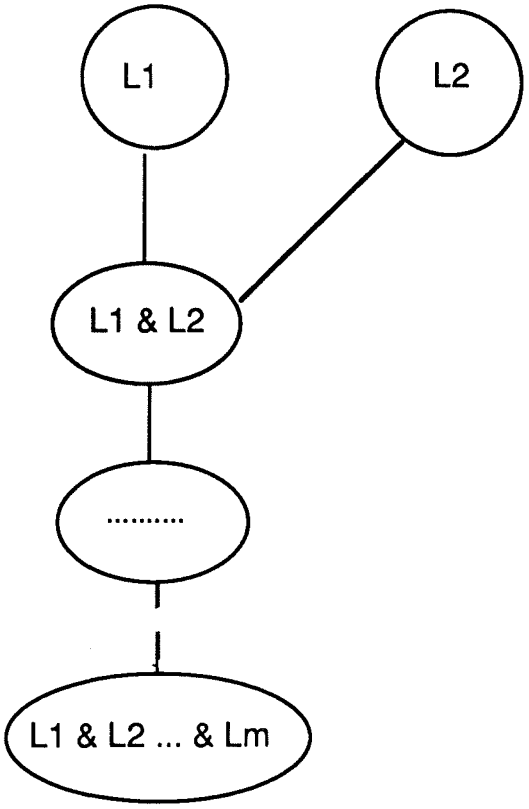
- générer : instructions de filtrage du littéral

RIGHT_MERGE L_i

Ainsi,en a) nous représentons



en b) nous complétons la séquence S et générons un noeud de filtrage



en c) nous terminons la séquence S et générons un noeud de filtrage

Quant à la partie droite, le résultat de sa compilation complétera S, se trouvant ainsi en séquence derrière le noeud terminal. On veillera simplement à traiter correctement les foncteurs spéciaux fail/0 et =/2 et à insérer une instruction SUCCEED_ALL en fin de séquence si le connecteur est ⇔.

3.4.3. Input et output du compilateur

En input, le compilateur a besoin d'une règle de production (mermbre de gauche, membre de droite, connecteur). Chaque règle de production est compilée séparément. Il est même possible de compiler des règles après le début de l'éxécution, le code généré s'ajoutant au code généré précédemment. Ceci donne du compilateur un certain caractère incrémental moyennant les valmidations appropriées, on pourrait peut-être développer un système véritablement incrémental. En output, le compilateur remet un code de diagnostic et le code généré (qui est privé : voir figure 1 au § 3.1.)

3.4.4. Les optimisations

Deux optimisations ont été apportées au schéma de compilation:

a) l'indexation sur un atomique 1er argument. A cette fin, nous rajoutons à la machine abstraite l'instruction switch_atomic :

SWITCH_ATOMIC : 2 arguments : arg_1 un enter m>0, arg_2 une table

a_1	c_1
a_2	c_2
.	.
.	.
.	.
a_m	c_m

avec ai un atomique et Ci l'adresse d'une instruction

Pré : pile d'adresses = adr_n ($n \geq 0$)

.

.

.

adr_1

tous les a_i sont distincts

Post : - si $\exists$ i : a_i = C (1), l'éxécution se poursuit en Ci

- sinon

* si n>0, pile d'adresses = adr_{n-1}

.

.

.

adr_1

et l éxécution se poursuit pour adr_n

* sinon, l'éxécution prend fin

b) la "factorisation" qui consiste à n'avoir qu'un seul noeud élémentaire pour toutes les occurences d'un même littéral (avec les mêmes arguments) dans toutes les règles de production. Le caractère incrémental du compilateur est préservé quoiqu'il faille alors éventuellement modifier le code déjà généré.

Exemple : soient les règles de production :

$x \geq y \;\; y \geq 0 \Rightarrow x \geq 0$

$y \geq x \;\; y = 0 \Leftrightarrow x = 0$

Le réseau de Rete correspondant est :

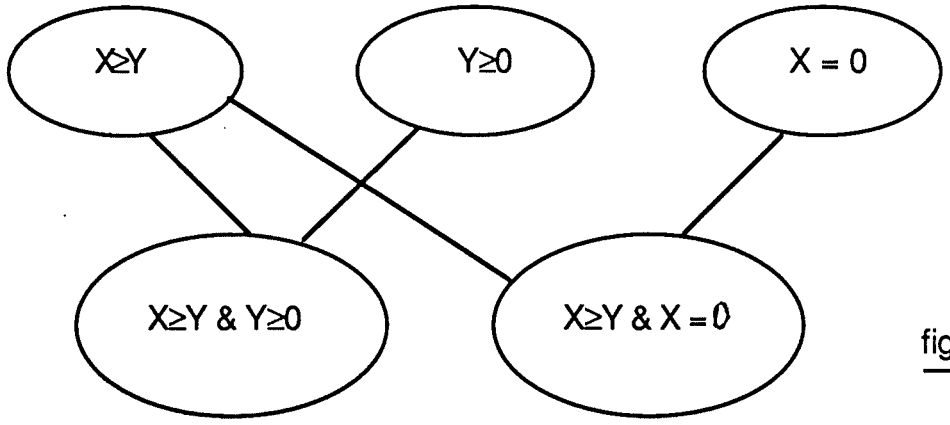


figure 7

(remarquez le chanfement de nom des variables)

Il y a donc 2 arcs à parcourir pour toute contrainte instan ant X≥Y d'où usage de l'instruction fork

3.4.5 Défauts du compilateur

a) Ce compilateur est lié à une topologie de réseau de Rete bien particulière : il n'autorise pas le choix d'une topologie par l'utilisateur. Or, la topologie choisie suppose que les contraintes les plus fortes sont au début du membre de gauche, minimisant ainsi le nombre de jointures réussies, donc le

nombre d'éléments à mémoriser dans les noeuds de jointure et par conséquent l'espace-mémoire requis à l'éxécution. Cette stratégie ne sied peut-être pas dans toutes les circonstances.

b) L'indexation (dont il est question au § 3.4.4., point a)) est limité au 1er argument. Une généralisation à n arguments (où n serait l'arité du littéral) serait probablement intéressante, diminuant le temps d'éxécution au détriment de la quantité d'espace-mémoire occupé par le code généré.

3.4.6. Exemple de compilation

Soit à compiler :

p (X,vrai,X) q(Y,X) q(x,Y,X) ⇔ X=5
p (_g406, vrai, _g406) ⇔ hello_world p(_g406, vrai,_g406) ⇔ true
q (a,a) ⇔ fail
q (a, Hobbit) ⇔ samwise_gamgee
q (a, Hobbit) ⇔ frodo_baggins
fib (X,Y) fib (X,Y) ⇔ fib (X,Y) q (a, Hobbit) ⇔ peregrin_took
q (a, Hobbit) ⇔ meriadoc_brandybuck

Le code généré est : (chaque fork est suivi du code généré qu'il adresse, indenté; les arguments du type "adresse d'instruction" n'apparaissent pas).

```
{ fib/2 : START }
FORK
        STORE_CSTR    7
        FORK
            LEFT_MERGE    7,7
            REMERGE
            TEST_EQ       (7,0), 0
            TEST_EQ       (7,0), 1
            MAKE_GOAL      fib/2
            PUSH_VALUE    7,0
            PUSH VALUE    7,1
            SUCCEED_ALL
            RETURN

        FORK
            RIGHT_MERGE    7,7

SWITCH_ATOMIC
{ fib/2 : } STOP

{ q/3 : START }
SWITCH_ATOMIC
X :
        FORK
            STORE_CSTR      3
            FORK
                RIGHT_MERGE    2,3
{ q/3 : } STOP

{ q/2 : START }
FORK
        STORE_CSTR    1
        FORK
            RIGHT_MERGE      0,1

SWITCH_ATOMIC
a:
```

```
            FORK
                STORE_CSTR      6
                FORK
                    MAKE GOAL    samwise_gamgee/0
                    RETURN
                FORK
                    MAKE_GOAL    frodo_baggins/0
                    RETURN
                FORK
                    MAKE_GOAL    peregrin_took/0
                    RETURN
                FORK
                    MAKE_GOAL    meriadoc_brandybuck/0
                    RETURN
                FORK
                    GET_ATOMIC   1,a
                    STORE_CSTR   5
                    FORK
                        FAIL

{ q/2 : } STOP

{ p/3 : START }
FORK
        GET_ATOMIC      1,vrai
        GET_VALUE       0,2
        STORE_CSTR      5
        FORK
            LEFT_MERGE  0,1
            REMERGE
            TEST_EQ         (0,0),1
            COMBINE         2
            LEFT_MERGE  2,3
            REMERGE
            TEST_EQ         (1,0), 1
            TEST_EQ         (0,0), 2
            UNIFY-ATOMIC        (0,0), 5
            SUCCEED_ALL
```

```
            RETURN
        FORK
            MAKE_GOAL           hello/1
            PUSH_ATOMIC         world
            RETURN
        FORK
            RETURN
SWITCH_ATOMIC
{ p/3 : } STOP

{ END_OF_CODE }
```

3.5. L'émulateur

Il s'agit simplement de l'adaptation à la représentation particulière du réseau de Rete (le code généré par le compilateur) de l'algorithme abstrait décrit au § 2.4.5. Les adaptations les plus importantes sont :

- l'implémentation des registres de la machine abstraite (voir § 3.3.2.3.), afin de pouvoir éxécuter l'une après l'autre les instructions de cette machine;

- la suppression de l'argument d'entrée N, décrit au § 2.4.5. représentant le réseau de Rete : celui-ci est mémorisé dans des zones de mémoire rémanentes (voir aussi la figure 1 au § 3.1.).

- l'application d'une règle applicable par l'éxécution des instructions résultant de la compilation du membre de droite de la règle (voir § 3.4.2.) et la production des résultats appropriés output.

- l'implémentation de la mise à jour de l'état d'instanciation des règles nécessité par la suppression implicite de contraintes (voir § 3.2., point C) : on se contente pour cela de vérifier lors de chaque jointure que chaque contrainte correspondant à chaque "ancien" élément d'un noeud existe toujours.

- la demande de suppression de contraintes victimes de l'application d'une règle de simplification, via une fonction fournie par l'interface (voir kill_g_ref () en annexe , point 3).

3.6. Adéquation au cahier des charges

Comme déjà mentionné au § 2.3.1., FR restreint la forme des littéraux du membre de gauche aux constantes, variables et structures à arguments simples. FR est probablement adaptable à tout compilateur Prolog et pourrait même être utilisé comme coeur d'un système autonome d'évaluation des règles de production.

CHAPITRE IV :
" A QUOI CA SERT ?"

FR n'a, à ma connaissance, jamais encore été utilisé. Néanmoins, il me semble que 3 types d'applications sont envisageables.

4.1. Système de réécriture

C'est dans ce sens que sont les exemples réalisés par Thomas Graf pour son implémentation des règles de production (5). Un simple exemple illustrera cette possibilité : soit à exprimer les connecteurs du calcul propositionnel : on pourrait simplifier oar exemple l'évaluation de l'expression and (Arg_1, vrai, Res) par la règle de simplification

$$\text{and (Arg1, vrai, Res)} \Leftrightarrow \text{Res} = Arg_1 ;$$

on pourrait ausi simplifier l'évaluation d'une double négation par la règle de déduction :

$$\text{not } (Arg_1, Res_1) \& \text{not } (Res_1, Res_2) \Leftrightarrow Arg_1 = Res_2$$

4.2. Méta-contrôle en Prolog

Une possibilité résultant du § 4.1. est le méta-contrôle de programmes Prolog. En ce cas, les contraintes sont simplement les termes Prolog présents dans la résolvante et les termes entrant dans la résolvante doivent être soumis à FR avant d'être examinés par le moteur d'inférence. Une possible application serait alors l'instauration d'une stratégie variable d'exploration de l'arbre de recherche. Esquissons un exemple : soit à explorer un labyrinthe, connaissant le point d'entrée, le point de sortie (distinct du point d'entrée) et le trajet déjà parcouru. Nous pourrions être amenés à élaguer l'arbre de recherche par la simplification :

$$\text{position (X,Y)} \& \text{enfermé (X,Y)} \Leftrightarrow \text{fail}$$

où position (X,Y) représente une coordonnée dans le labyrinthe et où enfermé (X,Y) est vrai si et seulement si la coordonnée (X,Y) se trouve dans un polygone délimité par les parois externes du labyrinthe et/ou par le trajet déjà parcouru, ce polygone ne contenant pas le point de sortie. En ce cas, il est inutile d'explorer ce polygone, on peut directement opérer un backtracking.

4.3. Applications de portfolio (7), (8)

Ces applications visent à la gestion d'un portefeuille boursier. Ce sont des systèmes experts, qui doivent tenir compte de 2 aspects différents du réel : les buts de l'utilisateur et la maximisation de ces buts. Une des approches

possibles, décrite (ou plutôt survolée) dans (7), est l'inférence des buts de l'utilisateur par un système de règles en chaînage avant complété par un ensemble de fonctions d'interviews et la maximisation des buts à atteindre par un algorithme d'inférence en chaînage arrière. Cette combinaison de chaînage avant et arrière est offerte par la combinaison de Prolog et de FR. On peut donc se demander si FR ne pourrait être utilisé dans un tel contexte.

BIBLIOGRAPHIE

(1) : Sterling, L. et Shapiro, E. The Art of Prolog : Advanced Programming Techniques. MIT Press (1986).

(2) : Van Henterijck, P. Constraint Satisfaction in Logic Programming. MIT Press (1986).

(3) : Cohen, J. Constraint Logic Programming Languages CommACM 33,7 (1990).

(4) : Colmerauer, A. An Intriduction to Prolog III CommACM 33,7 (1990).

(5) : Graf, T. Raisonner sur les contraintes en programmation logique. Thèse de doctorat, U. Nice (1989).

(6) : Forgy, C. Rete : A fast Algorithm for the Many Pattern / Many Object Pattern Match Problem . Artificial Intelligence 19 (1982).

(7) : Cohen, P. et Lieberman, M. A report on FOLIO : An expert assistant fort portfolio managers. IJCAI-83 vol.1.

(8) : Chan, Y., Saw, E. et Dellon, T. An expert system for a portfolio management using both frames and production rules. 8ème journée internationales d'Avignon vol.2 (1988).

ANNEXE

Annexe : Spécification de l'interface de FR

Cette annexe contient la version préliminaire d'un rapport interne destiné à l'ECRC.

Abstract

This report describes the interface to a forward rules compiler and emulator suitable for integration in a Prolog compiler.

1. Introduction

The interface described here must fulfill the following functionnalities :
- map the data structures used by the Prolog compiler with the structures used by the forward rules system;
- hide some information to the forward rules system by providing auxiliary functions;
- return results to Prolog according to the codes received from the forward rules system.

The interface must be made visible thru a header file called 'r_interface.h'.

2. Data Structures

2.1. Data types

The data types of the forward rules system are defined in the header files 'r_defines.h' and 'r_types.h'.

The relevant data types and auxiliary macros or functions of the Prolog compiler must be made visible thru a header file called 'external.h'.

2.2. Parameters of the compiler

The compiler is invoked separately for each source forward chaining clause.

The header of the compiler is :

```
Com_err     compiler (lit, size, operator)
Literal     lit[];
unsigned    size;
Operator    operator;
```

'lit' must be a pointer to an array of (size + 1) literals : lit [0] represents the RHS of the clause and lit [1]to lit [size] its LHS.

```
typedef struct
{
void     *id_proc,
         **arg;
}
         Literal;
```

'id_proc' points to some Prolog data structure identifying the procedure; each 'arg' is a pointer to an argument of the procedure. The number of *arg's is equal to the arity of the procedure.
The type 'Operator' is :

```
typedef enum
{
EQUIV,     /* correspond to the clause operator  ⇔   */
IMPLIES   /* correspond to the clause operator  ⇔   */
}
          Operator;
```

The type Com_err is:

```
typedef enum
{
OK,          /* compilation of the clause was without errors*/
```

```
U_ATOM,  /* compile error : attempt to unify 2 atomics in the RHS*/
U_FREE,   /* compile error : attempt to unify a free variable in the
RHS*/
IGNORED         /* the clause was ignored : see point 4*/
}
        Com_err;
```

The interface must check that there is no compound term among the arguments.

2.3. Parameters of the emulator

The emulator must be invoked for each literal received from the prolog system at run-time. Its header is :

```
Result          *emulator (g_ref, goal, size)
  void          *g_ref;
  Literal        goal;
  unsigned      *size;
```

where'g_ref' is a pointer to the Prolog goal,
'goal' is the goal after mapping as a 'literal',
'* size' is an uninitialized unsigned integrer.

The emulator returns a pointer to an array of 'Result' of size 'size'.
The type Result is defined as :

```
typedef enum
{
   SUCCESS,                /* no argument */
   FAILURE,                 /* no argument */
   UNIFY_A,                 /* 2 arguments */
   UNIFY_GOAL,              /* arity + 1 argument */
   DELAY                    /* 2 arguments */
}
      Return_code;

typedef union
{
  Return_code  return code
```

```
  void          *id_proc,
                *g_ref,
                *arg;
}
                Result;
```

For each 'Return_code' in the array, the interface must be take the appropriate action :

SUCCESS : return a 'succeed' result to the Prolog system;
FAILURE : return a 'fail' result to the Prolog system;
UNIFY_A : request the unification of the term pointed by the first argument with the atomic second argument;
UNIFY_V : request the unification of the terms pointed by the two arguments;
NEW_GOAL : request that a new goal be added to the resolvent : its procedure identifier is given by the first argument and its parameters by the others arguments;
DELAY : request that the Prolog goal pointed by the first argument be delayed ; it must be processed by the forward rules system when the variable second argument is unified.

3. The auxiliary functions

They may be implemented as C macros. These macros should not be terminated by a semi-colon.

```
unsigned        arity (id_proc)
  void          *id_proc;
```

Return the arity of the procedure identified by *id_proc.

```
char            is_true (id_proc)
  void          *id_proc;
```

Returns 1 (TRUE) if the procedure identified by *id_proc is 'true/0', 0 (FALSE) otherwise.

```
char            is_fail0 (id_proc)
```

```
  void          *id_proc;
```

Returns 1 (TRUE) if the procedure identified by *id_proc is 'fail/0, 0 (FALSE) otherwise.

```
char            is_unify2 (id_proc)
  void          *id_proc;
```

Returns 1 (TRUE) if the procedure identified by *id_proc is '=/2', 0 (FALSE) otherwise.

```
char            is_atomic (arg)
  void          *arg;
```

Returns 1 (TRUE) if * arg is an atomic, 0 (FALSE) otherwise.

```
char            is_free (arg)
  void          *arg;
```

Returns 1 (TRUE) if the variable pointed by 'arg' occurs for the first time in the clause, 0 (FALSE) otherwise.

```
char            is_ref (arg)
  void          *arg;
```

Returns 1 (TRUE) if, at compile time, the variable pointed by 'arg' has already occured in the LHS of the clause, 0 (FALSE) otherwise.

```
void            *copy_id_proc (id_proc)
  void          *id_proc;
```

Returns a pointer to a non-volatile copy (valid until the end of the execution) of *id_proc.

```
void            *copy_atom (arg)
  void          *arg;
```

Returns a pointer to a non-volatile copy of *arg.

```
char        same_atom (arg1, arg2)
  void      *arg1,*arg2;
```

Returns 1 (TRUE) if *arg 1 and *arg2 represent the same atomic,0 (FALSE) otherwise.

```
void        kill_gref (g_ref)
  void      *g_ref;
```

Requests that the prolog goal pointed by g_ref be deleted from the resolvent.

```
void            trail (loc)
 void           *loc;
```

Requests that the location loc be trailed.

4. Miscelleanous facilities useable by the Prolog system.

```
void            end_compile ()
```

Signals that no more clauses will be compiled. This allows the forward rules system to deallocate some dynamic arrays. Subsequently, any call to compiler() will return the code 'IGNORED'.

Réalisation, impression et reliures : SPRL Centre de Reprographie - Tél. : 081/22.87.72